International Journal of Trend in Scientific Research and Development (IJTSRD)
Volume 5 Issue 4, May-June 2021 Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

CSRF Attacks and its Defence using Middleware

Shubham Kumar Jha[1], Raghavendra R[2]

[1]MCA Student, [2]Assistant Professor, Department of MCA,
[1,2]School of CS & IT, Jain Deemed-to-be University, Bengaluru, Karnataka, India

ABSTRACT

A common solution to the issue of CSRF vulnerability is to restrict malicious requests from reaching the core of the application, where all the data and business logic is present. But the most challenging part is to identify when a request is malicious and when is it healthy. Implementing a simple solution would lead to more vulnerabilities and implementing too strict a solution would lead to breakages where projects depend on cross-site requests like third-party authentication and payment gateways etc.

The solution being proposed in this paper constitutes the design and implementation of a request-filtering mechanism that can precisely distinguish between malicious and healthy requests, and automatically decide to restrict them or allow them to get further deep into the system. This paper briefly explains what a Cross-Site-Request-Forgery attack is, and then goes into a step-by-step explanation on the prevention of CSRF attacks using a middleware. The proposed system is very strict in filtering out HTTP requests but also has an option to exempt certain cross-site requests based on their domain or URL, with which payment hooks and other third-party authentication calls can be exempted from the CSRF middleware.

KEYWORDS: *CSRF attack, cross-site request forgery, web security, Internet vulnerability*

How to cite this paper: Shubham Kumar Jha | Raghavendra R "CSRF Attacks and its Defence using Middleware" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-5 | Issue-4, June 2021, pp.1085-1088, URL: www.ijtsrd.com/papers/ijtsrd42476.pdf

IJTSRD42476

I. INTRODUCTION:

In today's plethora of software development technologies, websites and web servers are most commonly used. Even if we look at mobile applications, they are just the frontend skin of the whole product and it always requires a web server to handle all the business logic. And at this scale, servers are always prone to some or other security loopholes.

A very significant web vulnerability is Cross-site Request forgery. In a successful CSRF attack, the attacker causes the victim user to make a request unintentionally. It is a form of request hijacking. The request being made by a genuine user is hijacked and some malicious data is sent with the form data. In some cases, the attacker identifies the URL end-points of the server which are used for CRUD operations, and then sends a request to the identified end-point from a completely different origin, with a set of malicious form data. E.g., if the user has a session active in tab 'A' with a bank website, and has a malicious website open in tab 'B'. So, in a CSRF attack, the attacker on tab 'B' may identify the URL end-point for making a payment transaction on tab 'A' and perform a successful request on the user's behalf (without their knowledge) on the bank website and steal all the money from user's account [1].

CSRF has been recognized as a security threat for several years now and many authors have proposed various countermeasures including server-side filtering and client-side browser-based extensions. But with the browser-based or client-side approaches, there are major drawbacks.

- The attacker's request doesn't need to originate from the same browser, the attacker may use a completely different tool for making requests, and this can easily bypass browser extensions. So, server-side strict filtering is the best approach that can be taken, because it can be implemented easily as a central system that can filter all the requests irrespective of their origin.
- Client-side implementations are also very strict in terms of ceasing cross-origin site requests of a specific type (e.g., GET, POST, any), this eventually leads to breakages in existing websites and newer ones also which are dependent upon various valid and trusted cross-origin site requests (e.g., trusted 3rd party authentication, Payment Gateway hooks, etc.). This can be easily solved in a server-side filter because it has a centralized filtering system.

A. HTTP Request Methods:

Traditionally there have been only 2 HTTP request methods i.e., GET & POST, but with the evolution of web servers and their requirements and based on CRUD (Crud, Read, Update, Delete) operations, few new methods have come up named PUT, PATCH and DELETE. The newly added methods are developed by mimicking and augmenting the POST method only.

1. GET: The GET request method is used to request/fetch data from any specific resource present on the internet. It is the most commonly used HTTP request method.
2. POST: The POST method is used to send data to a web server to CREATE / UPDATE a resource.
3. PUT: The PUT method is used to send data to a web server to create/update a resource. The main difference

b/w POST and PUT are, calling the same PUT request multiple time will always produce the same result, but calling POST request multiple times will create the resource multiple times.

4. PATCH: The PATCH method is used to UPDATE a resource on any web server. The main difference b/w other updating methods and PATCH is, in PATCH only the data that needs to be updated is sent, without modifying affecting other data in the resource. [3]
5. DELETE: The DELETE method is used to remove/delete a resource on a web server identified by a web URI.

Most of the request methods excluding GET are concerned with data manipulation, and if an attacker gains access to any one of the methods, it could lead to disastrous ends. In most cases, POST methods are targeted by the attackers under CSRF attacks, to insert/update malicious data to benefit themselves. So, a strict filtering mechanism is needed to overcome this vulnerability. The system should be able to identify and stop any POST request which is not of a genuine origin or has malicious data. The proposed system in this paper does the same by implementing a middleware.

II. LITERATURE SURVEY

1. OWASP (Open Web Application Security Project) CSRF Guard is a mitigation strategy designed to protect against Cross-Site Request Forgery (CSRF) attacks. CSRF, also known as one-click attack or session riding, is one of the most dangerous threats against web applications. The consequence of successful CSRF exploit could result in disclosure of private information, unauthorized modification of sensitive data and disruption of web service.
2. A common client-side countermeasure against Cross Site Request Forgery (CSRF) is to strip session and authentication information from malicious requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures are typically too strict, thus breaking many existing websites that rely on authenticated cross-origin requests, such as sites that use third-party payment or single sign-on solutions.
3. Several applications extending the Hypertext Transfer Protocol (HTTP) require a feature to do partial resource modification. The existing HTTP PUT method only allows a complete replacement of a document. This proposal adds a new HTTP method, PATCH, to modify an existing HTTP resource.
4. For over a decade now, cross-site request forgery (CSRF) has been persistently named one of the OWASP's top 10 Web vulnerabilities. Recently, a variant of CSRF - named cross-site framing attack (CSFA) - has also been identified. Both attacks are very simple to implement/execute while resulting in potentially devastating consequences for the victim. What distinguishes the two attacks is their ultimate objective.
5. Cross-site request forgery (CSRF) vulnerability is extremely widespread and one of the top ten Web application vulnerabilities of the Open Web Application Security Project (OWASP).
6. As businesses are opening up to the web, securing their web applications becomes paramount. Nevertheless, the number of web application attacks is constantly increasing. Cross-Site Request Forgery (CSRF) is one of the more serious threats to web applications that gained a lot of attention lately. It allows an attacker to perform malicious authorized actions originating in the end-user's browser, without his knowledge.

III. PROBLEM STATEMENT

Cross-site request forgery (CSRF) is a very major security threat. It is considered one of the most common web vulnerabilities that exist today. In a successful CSRF attack, the attacker causes the victim user to make a request unintentionally. It is a form of request hijacking. The request being made by a genuine user is hijacked and some faulty data is sent with the form data to the original destination with malicious intent. (see Fig. 1)

E.g., Let's understand this using the banking scenario as an example.

1. The attacker creates a fake duplicate website of the target-bank.com at malicious-bank.com. And then sends the link for malicious-bank.com to the victim through email, social media, or SMS.
2. The victim sees the intriguing message and opens up the malicious-bank.com in their browser. When this page opens up it has a duplicate form similar to any actual form on the bank's website. But it already has a malicious payload in it.
3. Victim fills up their credentials and submits the form.
4. Then the request is sent to target-bank.com with the malicious payload and victims' credentials.
5. Target-bank.com accepts the request as it has the victims' actual credentials. With this, the attacker can even access the restricted parts of the application server and damage or take the data. Now the attacker can make payments on the victim's behalf and steal all his money and tinker with other vulnerabilities on target-bank.com.

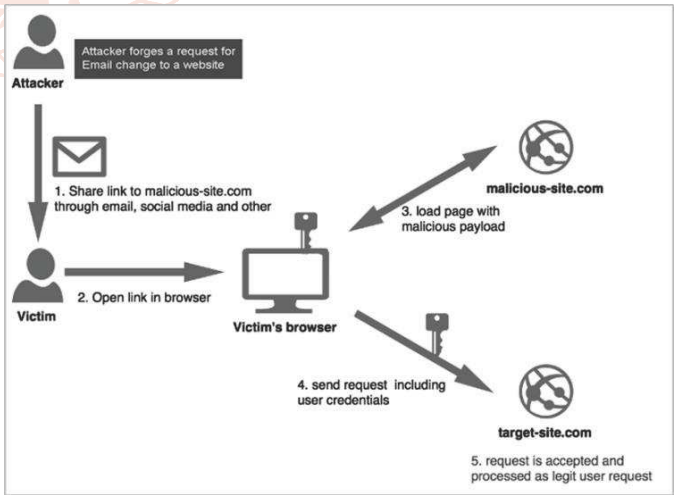

Fig 1 Example of a Successful CSRF Attack

IV. PROPOSED SYSTEM

There can be two approaches to this, one from the client-side and the other from the server-side. The client-side solutions are browser-specific and they monitor if any of the open tabs are trying to make a cross-origin request, and it simply restricts them. This method has many drawbacks like being too strict and lack of reverse compatibility for older projects.

On the other hand, this paper proposes a simple but effective server-side approach to fixing this vulnerability in web applications. In this, middleware is used to filter all of the POST requests being made on the server.

A. What is middleware?

A Middleware is a layer that wraps our application and its modules completely. And anything trying to access the restricted parts of the application has to go through this layer. This layer of middleware acts as a guard and makes the application more secure and safe.

On a technical level, Middleware is a piece of code, through which any request has to pass to access internal parts of an application (business logic, Database, User details, etc.)

B. Implementation:

- Generate a unique alphanumeric token of 40 characters which can be identified by the same servers only.
- Store the unique tokens with the users' sessions and refresh the tokens as the session changes.
- Render the token on every page from where an authentic request is expected via form or AJAX calls.
- Check every POST request for the token and match it with the one present in the users' session.
- If the token matches, allow the request to execute further.
- If the token is not found or it doesn't match, throw an HTTP 419 error.
- Token Example: "vVMDdtQKv8IHgF4rR7gq5V6BrjxYVzhhmp4VjX5G"
- See Fig. 2.

How this works is, the payload of every POST request is checked for the unique token generated by the server, if it is found valid, the request is allowed to persist or if the token is invalid or missing, the request is stopped from propagating further. This method precisely determines the origin of the request being made and successfully prevents a potential CSRF attack.

In this implementation, we also have the flexibility to allow certain cross-origin requests even while being a strict mechanism, as the whole filtering mechanism is in our hand rather than on the client's device. We can easily hardcode the verified domains and URLs into our middleware code which we want to exempt from the CSRF check.

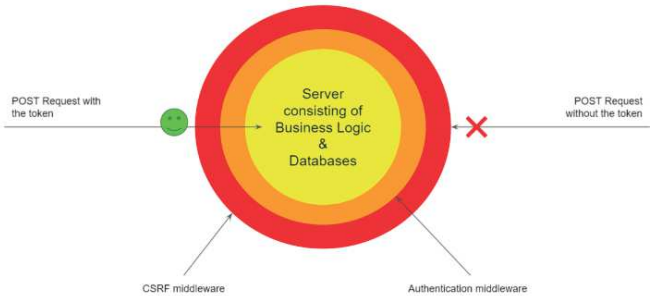

Fig 2 CSRF Middleware in Action

V. TEST CASES AND RESULTS

A. Test Case 1

1. Tool Used

- **Google Chrome.**

Google Chrome is a cross-platform web browser developed by Google. It was first released in 2008 for Microsoft Windows. It was later ported to Linux, macOS, iOS, and Android where it is the default browser built into the OS.

2. Steps

- When the markup of a page having a form is being generated on the server, the token is attached with the form and sent to the browser.
- So, when the user submits the form, the token is sent along with the form data and then it is matched with the token present in user session.
- If token matches, the request is allowed to make changes otherwise a HTTP 419 error is displayed and the request is stopped.
- This test case checks if the middleware allows the request from a genuine origin.
- Result of this was a pass, the user was allowed to continue the request as it had the token. See Fig. 3.

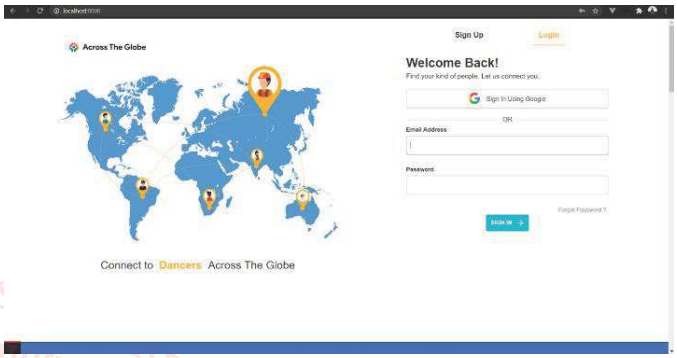

Fig 3 Test Case 1 Result - A Login Form from the same origin site

B. Test Case 2

1. Tool Used

- **Postman**

Postman is a collaboration platform for API development. Postman's features simplify each step of building an API and streamline collaboration so you can create better APIs—faster.

2. Steps

- Retrieve the target URL by inspecting the markup of the form, to which the form will make a request after submission. E.g., example.com/login.
- Make a POST request to the earlier identified URL, with the actual credentials of a user.
- The result of this is an HTTP 419 error because this payload doesn't have the token and the middleware blocks this from further execution. See Fig. 4.
- This test case confirms that the proposed solution is successful in blocking requests from another origin (postman in our case).

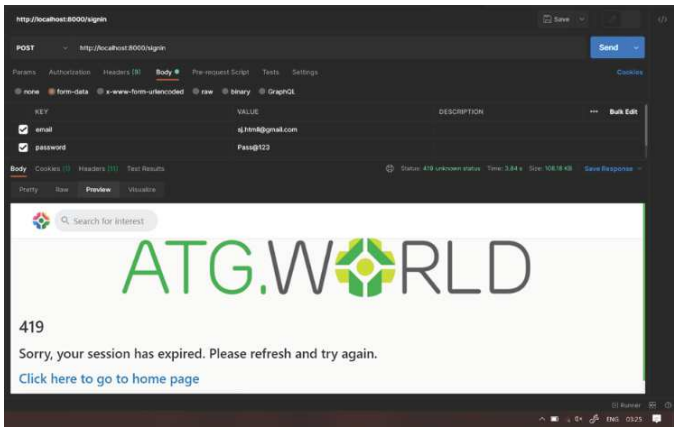

Fig 4 Test Case 2 Result - Attack on login form is successfully stopped by CSRF middleware

VI. CONCLUSION AND FUTURE WORK

CSRF is one of the top 10 common security threats present in any Web based product. There are many approaches available to counter this issue varying from client-side to server-side. The concept used in this paper is not new but the implementation of it using middleware is a valid and secure methodology. The approach of the server-side solution implemented in this paper to prevent CSRF attacks was found to be working as per requirement. It was also implemented in a live social media platform as a security patch. And as a result, it stopped many CSRF and Bot attacks.

Although this was found to be successful, it still needs a future study and research in the direction of preventing advanced bot attacks and request- overwhelming attacks, where an attacker designs a bot to keep sending large amount of malicious request to a server leading to a crash.

REFERENCES

[1] Chen, B., Zavarsky, P., Ruhl, R., & Lindskog, D. (2011). A Study of the Effectiveness of CSRF Guard. *IEEE*.

[2] De Ryck, P., Desmet, L., Joosen, W., & Piessens, F. (2011). Automatic and Precise Client-Side Protection. *Springer, Berlin, Heidelberg*.

[3] Dusseault, L., & Snell, J. (2010). PATCH Method for HTTP. *Internet Engineering Task Force (IETF)*.

[4] El Masari, M., & Vlajic, N. (2017). Current state of client-side extensions aimed at protecting against CSRF-like attacks. *IEEE*.

[5] Lin, X., Zavarsky, P., Ruhl, R., & Lindskog, D. (2009). Threat Modeling for CSRF Attacks. *IEEE*.

[6] Maes, W., Heyman, T., Desmet, L., & Joosen, W. (2009). Browser protection against cross-site request forgery. *SecuCode '09*.

[7] Mao, Z., Li, N., & Molloy, I. (2009). Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. *Springer, Berlin, Heidelberg*.

[8] Otwell, T. (n.d.). *Rotuing*. Retrieved from Laravel: https://laravel.com/docs/6.x/routing

[9] Rees, D. (2019). *Laravel Tutorial from Scratch for Beginners*. Retrieved from Morioh: https://morioh.com/p/9b8c8ef67bd5

[10] w3schools. (2018). *w3 schools*. Retrieved from HTTP Request Methods: https://www.w3schools.com/tags/ref_httpmethods.asp